

**Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores**

**Trabalho Prático 2**

47283 : Ricardo Duarte Cardoso Bernardino (a47283@alunos.isel.pt)

47249 : Miguel Henriques Couto de Almeida (a47249@alunos.isel.pt)

48270 : Daniel Lopes Pina (a48270@alunos.isel.pt)

Relatório para a Unidade Curricular de Inteligência Artificial  
da Licenciatura em Engenharia Informática e de Computadores

Professor : Doutor Nuno Miguel da Costa de Sousa Leite



# Contents

<b>List of Figures</b>	<b>v</b>
<b>Listagens</b>	<b>vii</b>
<b>1 Sudoku Solver</b>	<b>1</b>
1.1 Descrição do Problema . . . . .	1
1.2 Jogo . . . . .	1
1.2.1 Descrição do Jogo . . . . .	1
1.2.2 Funcionamento do Jogo . . . . .	2
1.3 Algoritmos . . . . .	3
1.3.1 <i>Deapth-First-Search Iterative Deepening</i> . . . . .	3
1.3.1.1 <i>Algoritmo Base</i> . . . . .	4
1.3.1.2 <i>Função Objetivo</i> . . . . .	4
1.3.1.3 <i>Função Sucessora</i> . . . . .	5
1.3.2 <i>A* Search</i> . . . . .	6
1.3.2.1 <i>Algoritmo Base</i> . . . . .	7
1.3.2.2 <i>Função Objetivo</i> . . . . .	8
1.3.2.3 <i>Função Sucessora</i> . . . . .	8
1.3.2.4 <i>Função Heurística</i> . . . . .	9
1.3.3 <i>Simulated Annealing</i> . . . . .	10

1.3.3.1	<i>Cálculo da Temperatura</i>	10
1.3.3.2	<i>Função de Custo</i>	11
1.3.3.3	<i>Algoritmo SA</i>	12
1.3.4	<i>Genetic Algorythm</i>	15
1.3.4.1	Seleção	15
1.3.4.2	Reprodução	16
1.3.4.3	Mutação	17
1.3.4.4	Solução Final	18
1.4	Conclusão	22
1.4.1	Resultados	22
1.4.2	Aprendizagem	22
	<b>Referências</b>	<b>25</b>

# List of Figures

1.1	Jogo em Curso . . . . .	2
1.2	Boards que foram usados nos Testes . . . . .	3
1.3	Resultados DFS-ID . . . . .	6
1.4	Resultados A* . . . . .	10
1.5	Resultados SA . . . . .	14
1.6	Resultados GA . . . . .	21
1.7	Resultados . . . . .	22



# Listagens

1.1	Algoritmo Base - DFS ID . . . . .	4
1.2	Função Objectivo - DFS ID . . . . .	4
1.3	Função Sucessora - DFS ID . . . . .	5
1.4	Algoritmo Base - A* Search . . . . .	7
1.5	Função Objectivo - A* Search . . . . .	8
1.6	Função Sucessora - A* Search . . . . .	9
1.7	Função Heurística - A* Search . . . . .	9
1.8	Cálculo da Temperatura - SA . . . . .	10
1.9	Função de Custo - SA . . . . .	11
1.10	Algoritmo SA . . . . .	12
1.11	Algoritmo de Selecção - GA . . . . .	15
1.12	Algoritmo de Reprodução - GA . . . . .	16
1.13	Algoritmo de Mutação - GA . . . . .	18
1.14	Algoritmo GA . . . . .	19







# Sudoku Solver

## 1.1 Descrição do Problema

Neste 2º trabalho prático foi proposto a implementássemos algoritmos de pesquisa informados e não informados com o propósito de resolver um jogo de tabuleiro chamado *Sudoku*[3]. Além dos algoritmos de pesquisa, também era proposto que lidássemos e entendêssemos o conceito de espaço de estados dentro do âmbito dos algoritmos propostos.

## 1.2 Jogo

### 1.2.1 Descrição do Jogo

*Sudoku* é um jogo jogado por 1 jogador numa grid de 9x9. O objetivo final é formar um tabuleiro completo sem que haja repetição de números na mesma linha, coluna e subgrid.

### 1.2.2 Funcionamento do Jogo

[illegible]

Figura 1.1: Jogo em Curso

## 1.3 Algoritmos

Dificuldade: Fácil Médio Difícil Especialista Diabólico

1					6	8	3	
	8		5	7	3		9	
				2	8			
	7		3		1	9	8	
5	4	9		8	7	6	1	
8			4		9	2		
					2	3		
7	6	3						9
9	2	8	7			5		

(a) Board Fácil usado nos Testes

Dificuldade: Fácil Médio Difícil Especialista Diabólico

5	3				4	2		
			9			7		4
				1	2	6		5
		4		6	3	1		
9		3		4	7		6	
6								2
	2	6	4		9	8		
	9	1	5					

(b) Board Médio usado nos Testes

Dificuldade: Fácil Médio Difícil Especialista Diabólico

6			7	9		2		8
						3		
	4		6					
	5				2	8		7
8							3	
				7			4	
4				2		6		9
		1			5			
							7	

(c) Board Diabólico usado nos Testes

Figura 1.2: Boards que foram usados nos Testes

### 1.3.1 Depth-First-Search Iterative Deepening

O algoritmo *Depth-First Search (DFS)* é um algoritmo de pesquisa em grafo que percorre este explorando o máximo possível em profundidade. Isto significa que o algoritmo visita o primeiro vizinho de um vértice antes de visitar o segundo vizinho e

assim sucessivamente, aprofundando-se na pesquisa até atingir um vértice sem filhos, momento esse em que volta a trás (backtracking) para visitar os outros vértices. O Iterative Deepening é uma variação deste algoritmo, para maximizar a eficiência de procura, para grafos com grandes profundidades, incrementando a profundidade gradualmente, e realizando várias chamadas ao algoritmo base DFS por profundidade. Só após explorar totalmente uma certa profundidade é que o algoritmo avança para uma nova.

#### 1.3.1.1 *Algoritmo Base*

---

```

1 :- consult('path.pl').
2
3 depth_first_iterative_deepening( Node, Solution) :-
4   path( Node, GoalNode, Solution),
5   goal( GoalNode).
6
7
8 % ?- depth_first_iterative_deepening(a, Solution).
9
10
11 path(Node, Node, [Node]). % Single node path
12
13 path(FirstName, LastName, [LastName | Path]) :-
14   path(FirstName, OneButLast, Path), % Path up to one-but-last node
15   s(OneButLast, LastName),          % Last step
16   \+ member(LastNode, Path).        % No cycle
17
18
19 % ?- path(a, Last, Path).
```

---

Listagem 1.1: Algoritmo Base - DFS ID

#### 1.3.1.2 *Função Objetivo*

A função objetivo averigua apenas se existem posições vazias no tabuleiro, sendo que, se existirem, ainda não foi concluída a resolução deste. Ao não existirem posições vazias no tabuleiro, é assim atingido o objetivo final do algoritmo, e concluído que todas as posições foram devidamente preenchidas.

---

```

1 goal(Board) :-
```

---

```
2  empty_positions(Board, List),
3  List == [].
```

---

Listagem 1.2: Função Objectivo - DFS ID

### 1.3.1.3 Função Sucessora

A função sucessora é implementada através da simples lógica de seleccionar a primeira posição vazia do tabuleiro, e gerar um número aleatório entre 1 e 9, visto que são as únicas possibilidades. Após a geração do número aleatório, é averiguada a presença de erros desta inserção, tentando validar a jogada realizada, falhando caso hajam.

---

```
1  s(Board, NextBoard) :-
2    first_empty_position(Board, X/Y),
3    member(Num, [1,2,3,4,5,6,7,8,9]),
4    validate_move(Board, X/Y, Num, NextBoard),
5    write(" [DEBUG] -> "), writeln(NextBoard).
```

---

Listagem 1.3: Função Sucessora - DFS ID

```

[DEBUG] -> [[1,5,7,9,4,6,8,3,2],[2,8,4,5,7,3,1,9,6],
,4,5,6,-]]
[DEBUG] -> [[1,5,7,9,4,6,8,3,2],[2,8,4,5,7,3,1,9,6],
,4,5,6,1]]
-----
| 1 5 7 | 9 4 6 | 8 3 2 |
| 2 8 4 | 5 7 3 | 1 9 6 |
| 3 9 6 | 1 2 8 | 7 4 5 |
-----
| 6 7 2 | 3 5 1 | 9 8 4 |
| 5 4 9 | 2 8 7 | 6 1 3 |
| 8 3 1 | 4 6 9 | 2 5 7 |
-----
| 4 1 5 | 6 9 2 | 3 7 8 |
| 7 6 3 | 8 1 5 | 4 2 9 |
| 9 2 8 | 7 3 4 | 5 6 1 |
-----
Execution time= 2443
true █

```

(a) Fácil

```

[DEBUG] -> [[5,3,8,6,7,4,2,1,9],[1,6,2,9,5,8,7,3,4],[4,7,9,3,1,2,6,8,5],
,6,4,7,-]]
[DEBUG] -> [[5,3,8,6,7,4,2,1,9],[1,6,2,9,5,8,7,3,4],[4,7,9,3,1,2,6,8,5],
,6,4,7,-]]
[DEBUG] -> [[5,3,8,6,7,4,2,1,9],[1,6,2,9,5,8,7,3,4],[4,7,9,3,1,2,6,8,5],
,6,4,7,-]]
[DEBUG] -> [[5,3,8,6,7,4,2,1,9],[1,6,2,9,5,8,7,3,4],[4,7,9,3,1,2,6,8,5],
,6,4,7,3]]
-----
| 5 3 8 | 6 7 4 | 2 1 9 |
| 1 6 2 | 9 5 8 | 7 3 4 |
| 4 7 9 | 3 1 2 | 6 8 5 |
-----
| 2 5 4 | 8 6 3 | 1 9 7 |
| 9 1 3 | 2 4 7 | 5 6 8 |
| 6 8 7 | 1 9 5 | 3 4 2 |
-----
| 3 4 5 | 7 8 1 | 9 2 6 |
| 7 2 6 | 4 3 9 | 8 5 1 |
| 8 9 1 | 5 2 6 | 4 7 3 |
-----
Execution time= 45961
true █

```

(b) Média

```

[DEBUG] -> [[6,1,3,7,9,4,2,5,8],[9,8,7,2,5,1,3,6,4],
[DEBUG] -> [[6,1,3,7,9,4,2,5,8],[9,8,7,2,5,1,3,6,4],
-----
| 6 1 3 | 7 9 4 | 2 5 8 |
| 9 8 7 | 2 5 1 | 3 6 4 |
| 5 4 2 | 6 8 3 | 7 9 1 |
-----
| 3 5 6 | 9 4 2 | 8 1 7 |
| 8 7 4 | 5 1 6 | 9 3 2 |
| 1 2 9 | 3 7 8 | 5 4 6 |
-----
| 4 3 5 | 1 2 7 | 6 8 9 |
| 7 9 1 | 8 6 5 | 4 2 3 |
| 2 6 8 | 4 3 9 | 1 7 5 |
-----
Execution time= 2720313
true █

```

(c) Diabólico

Figura 1.3: Resultados DFS-ID

### 1.3.2 A\* Search

A\* é um algoritmo de pesquisa em grafos ponderados, partindo de um nó inicial específico de um grafo, ele visa encontrar um caminho para o *goal* dado com o menor custo. O algoritmo mantém uma árvore de caminhos originados no nó inicial e estende esses caminhos aresta a aresta aprofundando a pesquisa até atingir um nó terminal (sem filhos), momento esse em que volta a trás (backtracking) para visitar os outros nós, combinando o custo acumulado até o momento (representado pela função " $g(n)$ ") com a heurística estimada (representada pela função " $h(n)$ "), o A\* é capaz de selecionar nós terminais que têm o custo total mais baixo (representado por " $f(n) = g(n) + h(n)$ "). Isso permite que o algoritmo explore os caminhos mais promissores primeiro, em vez de seguir uma busca cega em todas as direções.

**1.3.2.1 Algoritmo Base**


---

```

1  bestfirst(Start, Solution) :-
2    expand([], l(Start, 0/0), 9999, _, yes, Solution). % Assume 9999 is > any f-value
3
4  expand(P, l( N, _), _, _, yes, [N | P]) :-
5    goal(N).
6
7  expand(P, l(N, F/G), Bound, Tree1, Solved, Sol) :-
8    F <= Bound,
9    ( bagof( M/C, ( s(N, M, C), \+ member(M, P)), Succ),
10     !,
11     succlist( G, Succ, Ts),
12     bestf( Ts, F1),
13     expand( P, t(N, F1/G, Ts), Bound, Tree1, Solved, Sol)
14     ;
15     Solved = never
16     ).
17
18  expand( P, t(N, F/G, [T | Ts]), Bound, Tree1, Solved, Sol) :-
19    F <= Bound,
20    bestf(Ts, BF), Bound1 = min( Bound, BF),
21    expand( [N | P], T, Bound1, T1, Solved1, Sol),
22    continue( P, t(N, F/G, [T1 | Ts]), Bound, Tree1, Solved1, Solved, Sol).
23
24  expand( _, t( _, _ , []), _, _, never, _ ) :- !.
25  expand( _, Tree, Bound, Tree, no, _ ) :-
26    f( Tree, F), F > Bound.
27
28  continue( _,_,_,_, yes, yes, Sol).
29  continue( P, t(N, F/G, [T1 | Ts]), Bound, Tree1, no, Solved, Sol) :-
30    insert( T1, Ts, NTs),
31    bestf( NTs, F1),
32    expand( P, t(N, F1/G, NTs), Bound, Tree1, Solved, Sol).
33  continue( P, t(N, F/G, [_ | Ts]), Bound, Tree1, never, Solved, Sol) :-
34    bestf( Ts, F1),
35    expand( P, t(N, F1/G, Ts), Bound, Tree1, Solved, Sol).
36
37  succlist( _, [], []).

```

---

```

38 succlist( GO, [N/C | NCs], Ts) :-
39   G is GO + C,
40   h( N, H),
41   F is G + H,
42   succlist( GO, NCs, Ts1),
43   insert( l(N, F/G), Ts1, Ts).
44
45 insert( T, Ts, [T | Ts]) :-
46   f( T, F), bestf( Ts, F1),
47   F =< F1, !.
48 insert( T, [T1 | Ts], [T1 | Ts1]) :-
49   insert( T, Ts, Ts1).
50
51 f( l( _, F/ _), F).
52 f( t( _, F/ _ _), F).
53
54 bestf( [T | _], F) :-
55   f( T, F).
56 bestf( [], 9999).
```

---

Listagem 1.4: Algoritmo Base - A\* Search

### 1.3.2.2 Função Objetivo

A função objetivo averigua apenas se existem posições vazias no tabuleiro, sendo que, se existirem, ainda não foi concluída a resolução deste. Ao não existirem posições vazias no tabuleiro, é assim atingido o objetivo final do algoritmo, e concluído que todas as posições foram devidamente preenchidas.

---

```

1 goal(Board) :-
2   empty_positions(Board, List),
3   List == [].
```

---

Listagem 1.5: Função Objetivo - A\* Search

### 1.3.2.3 Função Sucessora

A função sucessora é implementada através da simples lógica de selecionar a primeira posição vazia do tabuleiro, e gerar um número aleatório entre 1 e 9, visto que são as



únicas possibilidades. Após a geração do número aleatório, é averiguada a presença de erros desta inserção, tentando validar a jogada realizada, falhando caso hajam.

---

```
1 s(Board, NextBoard) :-  
2   first_empty_position(Board, X/Y),  
3   member(Num, [1,2,3,4,5,6,7,8,9]),  
4   validate_move(Board, X/Y, Num, NextBoard),  
5   write(" [DEBUG] -> "), writeln(NextBoard).
```

---

Listagem 1.6: Função Sucessora - A\* Search

#### 1.3.2.4 Função Heurística

[2][1] A heurística é usada para orientar a busca do A\* e influencia a seleção dos nós. Ela fornece uma indicação de quão promissores são os nós não explorados em termos de proximidade do objetivo. Ao utilizar essa informação, o algoritmo pode priorizar a exploração de nós que têm uma heurística mais baixa, ou seja, que parecem estar mais próximos do objetivo. No caso do sudoku irá priorizar nós com o menor numero de espaços vazios possíveis.

---

```
1 h(Board, H) :-  
2   empty_positions(Board, List),  
3   length(List, H).
```

---

Listagem 1.7: Função Heurística - A\* Search

```
[DEBUG] -> [[1,5,7,9,4,6,8,3,2],[2,8,4,5,7,3,1,9,6]]
[DEBUG] -> [[1,5,7,9,4,6,8,3,2],[2,8,4,5,7,3,1,9,6]]

| 1 5 7 | 9 4 6 | 8 3 2 |
| 2 8 4 | 5 7 3 | 1 9 6 |
| 3 9 6 | 1 2 8 | 7 4 5 |
-----
| 6 7 2 | 3 5 1 | 9 8 4 |
| 5 4 9 | 2 8 7 | 6 1 3 |
| 8 3 1 | 4 6 9 | 2 5 7 |
-----
| 4 1 5 | 6 9 2 | 3 7 8 |
| 7 6 3 | 8 1 5 | 4 2 9 |
| 9 2 8 | 7 3 4 | 5 6 1 |
-----

Execution time= 36
true
```

(a) Fácil

```
[DEBUG] -> [[5,3,8,6,7,4,2,1,9],[1,6,2,9,5,8,7,3,4]]
[DEBUG] -> [[5,3,8,6,7,4,2,1,9],[1,6,2,9,5,8,7,3,4]]

| 5 3 8 | 6 7 4 | 2 1 9 |
| 1 6 2 | 9 5 8 | 7 3 4 |
| 4 7 9 | 3 1 2 | 6 8 5 |
-----
| 2 5 4 | 8 6 3 | 1 9 7 |
| 9 1 3 | 2 4 7 | 5 6 8 |
| 6 8 7 | 1 9 5 | 3 4 2 |
-----
| 3 4 5 | 7 8 1 | 9 2 6 |
| 7 2 6 | 4 3 9 | 8 5 1 |
| 8 9 1 | 5 2 6 | 4 7 3 |
-----

Execution time= 40
true
```

(b) Média

```
[DEBUG] -> [[6,1,3,7,9,4,2,5,8],[9,8,7,2,5,1,3,6,4]]
[DEBUG] -> [[6,1,3,7,9,4,2,5,8],[9,8,7,2,5,1,3,6,4]]

| 6 1 3 | 7 9 4 | 2 5 8 |
| 9 8 7 | 2 5 1 | 3 6 4 |
| 5 4 2 | 6 8 3 | 7 9 1 |
-----
| 3 5 6 | 9 4 2 | 8 1 7 |
| 8 7 4 | 5 1 6 | 9 3 2 |
| 1 2 9 | 3 7 8 | 5 4 6 |
-----
| 4 3 5 | 1 2 7 | 6 8 9 |
| 7 9 1 | 8 6 5 | 4 2 3 |
| 2 6 8 | 4 3 9 | 1 7 5 |
-----

Execution time= 710848
true [print]
```

(c) Diabólico

Figura 1.4: Resultados A\*

### 1.3.3 Simulated Annealing

#### 1.3.3.1 Cálculo da Temperatura

```
1 fun calculateStandardDeviation(numbers: List<Double>): Double {
2     val mean = numbers.average()
3     val variance = numbers.map { (it - mean) * (it - mean) }.average()
4     return kotlin.math.sqrt(variance)
5 }
6
7 fun calculateStartingTemperature(template: String): Double {
8     val results = mutableListOf<Int>()
9     val board = getBoardFromTemplate(template)
10    repeat(200) {
11        val filledBoard = board.fillWithRandValues()
12        results.add(filledBoard.getTotalCost())
13    }
```

```
14  return calculateStandardDeviation(results.map { it.toDouble() })
15 }
```

---

#### Listagem 1.8: Cálculo da Temperatura - SA

Esta função calcula a temperatura inicial para o algoritmo Simulated Annealing gerando aleatoriamente uma série de tabuleiros preenchidos com valores aleatórios com base no modelo fornecido. De seguida, calcula o custo total de cada tabuleiro gerado e armazena os resultados numa lista. Finalmente, retorna o desvio padrão dos resultados para ser usado como temperatura inicial na função de minimização.

##### 1.3.3.2 Função de Custo

---

```
1  fun getTotalCost(): Int {
2      var rows = 0
3      var cols = 0
4      for (index in 0 until BOARD_SIDE) {
5          rows += getErrorsRow(index)
6          cols += getErrorsColumn(index)
7      }
8      return rows + cols
9  }
```

---

#### Listagem 1.9: Função de Custo - SA

Esta função calcula o custo total de um tabuleiro que no contexto do algoritmo Simulated Annealing representa a medida que determina o quão bom é o tabuleiro atual em relação à solução desejada. Quanto maior o custo total, pior é o tabuleiro visto que estamos a tentar minimizar a temperatura.

A função `getTotalCost()` itera sobre todas as linhas e colunas do tabuleiro e calcula o número de erros em cada uma. Ela chama as funções `getErrorsRow(index)` e `getErrorsColumn(index)` para obter o número de erros em cada linha e coluna, respetivamente. Essas funções retornam a contagem de erros para uma determinada linha ou coluna.

Ao percorrer todas as linhas e colunas, a função acumula os números de erros em `rows` e `cols`. Por fim, retorna a soma de `rows` e `cols`, que representa o custo total do tabuleiro.

O custo total é crucial no algoritmo Simulated Annealing, pois é usado para avaliar a qualidade das soluções durante o processo otimização.

**1.3.3.3 Algoritmo SA**


---

```

1 fun SA(template: String): Board {
2     val initialBoard = getBoardFromTemplate(template)
3     initialBoard.print()
4
5     var currentBoard = initialBoard.fillWithRandValues()
6 2   var currentCost = currentBoard.getTotalCost()
7
8     var temperature = calculateStartingTemperature(template)
9
10    var stuckCounter = 0
11    var previousCost = 0
12
13    while (currentCost > 0 && temperature > 0.0) {
14        previousCost = currentCost
15        for (iter in 1..ITERATIONS_PER_TEMPERATURE) {
16            println(currentCost)
17            val newBoard = currentBoard.generateNextBoard()
18            val newCost = newBoard.getTotalCost()
19            val diff = newCost - currentCost
20
21            if (newCost < currentCost) {
22                currentBoard = newBoard
23                currentCost = newCost
24            } else if (exp(-diff / temperature) > Random.nextDouble()) {
25                currentBoard = newBoard
26                currentCost = newCost
27            }
28        }
29
30        if (currentCost >= previousCost) {
31            stuckCounter += 1
32        } else {
33            stuckCounter = 0
34        }
35        if (stuckCounter > 100){
36            //re-heat
37            temperature += 2

```

```
38     }  
39  
40     temperature *= COOLING_RATE  
41 }  
42 return currentBoard  
43 }
```

---

#### Listagem 1.10: Algoritmo SA

Esta função implementa o algoritmo Simulated Annealing. Recebe um Board no formato String como argumento e retorna um Board completo no final. O algoritmo começa por transformar a String num tabuleiro que possa ser usado. De seguida, o tabuleiro é preenchido com valores aleatórios e é calculado custo total do tabuleiro.

O algoritmo continua iterando enquanto o custo total não atingir zero e a temperatura for maior que zero. Durante cada iteração, um novo tabuleiro é gerado a partir do tabuleiro atual. O custo total do novo tabuleiro é calculado e comparado com o custo do tabuleiro atual e se o novo custo for menor, o tabuleiro atual é substituído pelo novo tabuleiro, caso contrário, o novo tabuleiro pode ser aceite com uma certa probabilidade, determinada pela função exponencial, pois é essa probabilidade que permite aceitar movimentos que aumentem o custo total, mas que também permitam escapar de mínimos locais.

O algoritmo também se o custo total permanece o mesmo por um certo número de iterações consecutivas (contador de "stuck"). Se isso ocorrer, o algoritmo aumenta a temperatura para "reiniciar" e permitir maior exploração.

A temperatura vai sendo reduzida de acordo com uma taxa de arrefecimento e o processo continua até que o custo total atinja zero ou que a temperatura caia para zero. Com temperatura a zero e a função de custo minimizada atingimos um board finalizado ou resolvido, que é o caso do nosso problema.

```

0
0
+-----+-----+-----+
| 1 5 7 | 9 4 6 | 8 3 2 |
| 2 8 4 | 5 7 3 | 1 9 6 |
| 3 9 6 | 1 2 8 | 7 4 5 |
+-----+-----+-----+
| 6 7 2 | 3 5 1 | 9 8 4 |
| 5 4 9 | 2 8 7 | 6 1 3 |
| 8 3 1 | 4 6 9 | 2 5 7 |
+-----+-----+-----+
| 4 1 5 | 6 9 2 | 3 7 8 |
| 7 6 3 | 8 1 5 | 4 2 9 |
| 9 2 8 | 7 3 4 | 5 6 1 |
+-----+-----+-----+
Execution time = 1406 ms

Process finished with exit code 0

```

(a) Fácil

```

0
0
0
+-----+-----+-----+
| 5 3 8 | 6 7 4 | 2 1 9 |
| 1 6 2 | 9 5 8 | 7 3 4 |
| 4 7 9 | 3 1 2 | 6 8 5 |
+-----+-----+-----+
| 2 5 4 | 8 6 3 | 1 9 7 |
| 9 1 3 | 2 4 7 | 5 6 8 |
| 6 8 7 | 1 9 5 | 3 4 2 |
+-----+-----+-----+
| 3 4 5 | 7 8 1 | 9 2 6 |
| 7 2 6 | 4 3 9 | 8 5 1 |
| 8 9 1 | 5 2 6 | 4 7 3 |
+-----+-----+-----+
Execution time = 3677 ms

Process finished with exit code 0

```

(b) Média

```

0
0
+-----+-----+-----+
| 6 1 3 | 7 9 4 | 2 5 8 |
| 9 8 7 | 2 5 1 | 3 6 4 |
| 5 4 2 | 6 8 3 | 7 9 1 |
+-----+-----+-----+
| 3 5 6 | 9 4 2 | 8 1 7 |
| 8 7 4 | 5 1 6 | 9 3 2 |
| 1 2 9 | 3 7 8 | 5 4 6 |
+-----+-----+-----+
| 4 3 5 | 1 2 7 | 6 8 9 |
| 7 9 1 | 8 6 5 | 4 2 3 |
| 2 6 8 | 4 3 9 | 1 7 5 |
+-----+-----+-----+
Execution time = 76464 ms

Process finished with exit code 0

```

(c) Diabólico

Figura 1.5: Resultados SA

### 1.3.4 Genetic Algorithm

Os Algoritmos Genéticos são inspirados no princípio Darwiniano da evolução das espécies e na genética. São algoritmos probabilísticos que fornecem um mecanismo de busca paralela e adaptativa baseado no princípio de sobrevivência dos mais aptos e na reprodução. Neste sentido, a implementação do *Genetic-Algorithm* para efeitos de resolução de um jogo de Sudoku, é puramente baseado nos seus princípios fundamentais, que são adaptados às regras do jogo de sudoku, ou seja a seleção, seguida da reprodução, seguida da mutação.

#### 1.3.4.1 Seleção

A seleção dos pais pode ser realizada de diferentes formas, porém a abordagem decidida foi através do modo de torneio. O modo de torneio, ou *tourney*, seleciona  $X$  indivíduos, e compete-os uns contra os outros baseado-se nos seus scores, ou seja, do seu valor de fitness. No caso do jogo de sudoku, um maior score, indica um maior valor de fitness, pois o score indica o número de casas do tabuleiro que não esteja em conflito, ou seja, em erro, no tabuleiro em questão, sendo que um tabuleiro com menos erros é um indivíduo mais desejável. Neste torneio, são selecionados 5 indivíduos, e retornado aquele que tiver um maior score. Foi também implementada a seleção por roleta, selecionando indivíduos com probabilidades proporcionais ao seu score.

---

```

1
2  /**
3   * Seleção por Roleta. Seleciona um individuo com uma
      probabilidade proporcional ao seu score.
4   */
5  fun <T> fitnessProportionateSelection(scoredPopulation: Collection<Pair<Double, T
      >>): T {
6      var value= scoredPopulation.sumOf { it.first } * random()
7
8      for ((fitness, individual) in scoredPopulation) {
9          value -= fitness
10         if (value <= 0) return individual
11     }
12
13     return scoredPopulation.last().second
14 }
15
16 /**

```

---

```

17  * Modo de seleção por Torneio. Seleciona 5 indivíduos randomly
    da população, e retorna aquele com maior score
18  */
19  fun <T> sudokuSelectionTournament(scoredPopulation: Collection<Pair<Double, T
    >>): T {
20      val tournamentSize = 5
21
22      val tournament = scoredPopulation.shuffled().take(tournamentSize)
23      val winner = tournament.maxByOrNull { it.first }!!.second
24
25      return winner
26  }

```

---

Listagem 1.11: Algoritmo de Seleção - GA

#### 1.3.4.2 Reprodução

A Reprodução, para criação de filhos, através de dois pais, é realizada nesta implementação através da criação de novos tabuleiros (filhos), através da junção genética de dois tabuleiros diferentes (pais). A junção genética, ou crossover, é realizada pela função *sudokuCrossover* apresentada abaixo. Esta função gera um número aleatório entre 1 a 7, representando um índice de bloco do tabuleiro, para que haja uma divisão justa, não é gerado um índice em que nenhum bloco terá sido selecionado de um dos pais, ou seja, no mínimo (índice gerado é igual a 7, ou seja o 8º bloco) é sempre selecionado 1 bloco de um dos pais. De seguida, é retirado bloco a bloco, até atingir esse índice, o/os bloco/os do pai, e finalmente após esse índice, é retirado o/os bloco/os da mãe. A junção final será o filho (tabuleiro) gerado.

---

```

1  /**
2   * Faz o crossover entre 2 parentes, para gerar um filho. O
    filho resultante terá os primeiros n blocos do parent1,
3   * até um crosspoint gerado randomly, e os restantes blocos do
    parent2.
4   */
5   fun sudokuCrossover(parents: Pair<List<List<Cell>>, List<List<Cell>>>): List<List<
    Cell>> {
6       val crossoverPoint = (0..7).random()
7       val child: MutableList<MutableList<Cell>> = mutableListOfOf()
8

```



```
9  for (i in 0 until 9) {
10     val row: MutableList<Cell> = mutableListOf()
11     for (j in 0 until 9) {
12         val cell = Cell(0) // Initialize every cell with the digit 0
13         row.add(cell)
14     }
15     child.add(row)
16 }
17 println("Parent 1: ")
18 Board(parents.first).print()
19
20 println("Parent 2: ")
21 Board(parents.second).print()
22 var childBoard = Board(child)
23
24 var blockPositions : List<Pair<Int,Position>>
25 var block : List<Cell>
26
27 val parent1 = Board(parents.first)
28 val parent2 = Board(parents.second)
29
30 for(i in 0 .. crossoverPoint){
31     block = parent1.getBlock(i)
32     blockPositions = parent1.getBlockPositions(i).toList()
33     println(blockPositions.size)
34     blockPositions.forEach {
35         childBoard = childBoard.insertCell(it.second,block[i])
36     }
37 }
```

---

Listagem 1.12: Algoritmo de Reprodução - GA

### 1.3.4.3 Mutação

A mutação dos filhos é realizada através da função *sudokuMutation* que seleciona aleatoriamente 1 bloco do tabuleiro desse filho. Neste bloco, como é sempre garantida a regra de não repetição de números dentro desta, é trocado aleatoriamente um número que não seja fixo, ou seja, que não tenha sido gerado pelo algoritmo, mas sim estava presente no estado inicial do tabuleiro dado ao algoritmo.

---

```

1  /**
2   * Seleciona randomly 1 bloco da board, e depois seleciona
      randomly 2 números não-fixos, que serão trocados um com o
      outro.
3   */
4  fun sudokuMutation(individual: List<List<Cell>>) : List<List<Cell>> {
5      val blockIndex = (0 .. 8).random()
6      val board = Board(individual)
7      val blockPositions = board.getBlockPositions(blockIndex).toList()
8      /**Tenta encontrar um valor desse bloco que não seja fixo**/
9      var randomNum1 = (0 .. 8).random()
10     var num1 = blockPositions[randomNum1]
11     var value1 = board.get(num1.second)
12     while(value1.fixed){
13         randomNum1 = (0 .. 8).random()
14         num1 = blockPositions[randomNum1]
15         value1 = board.get(num1.second)
16     }
17     /**Tenta encontrar um valor desse bloco que não seja fixo**/
18     var randomNum2 = (0 .. 8).random()
19     var num2 = blockPositions[randomNum2]
20     var value2 = board.get(num2.second)
21     while(value2.fixed || value2 == value1){
22         randomNum2 = (0 .. 8).random()
23         num2 = blockPositions[randomNum2]
24         value2 = board.get(num2.second)
25     }
26
27     /**Efetua a troca**/
28     return board.insert(num1.second,num2.first).insert(num2.second,num1.first).board
29 }

```

---

Listagem 1.13: Algoritmo de Mutação - GA

#### 1.3.4.4 Solução Final

Agora com as 3 funções principais do *Genetic Algorithm* realizadas, com base num jogo de sudoku, a função principal do algoritmo é implementada com uma função *Run* que

para 1000 epochs, ou seja 1000 ciclos, realiza o processo conjunto de seleção, reprodução e mutação, a cada ciclo, sob a população apresentada. Após estes ciclos de evolução genética, é esperado, sob uma certa probabilidade elevada, que o tabuleiro, ou filho, retornado por esta função, seja um que tenha um score total de 81, ou seja contenha 81 casas sem conflitos, um tabuleiro resolvido, sem erros.

```

1  class GA<T>(  

2      var population: Collection<T>,  

3      val score: (individual: T) -> Double,  

4      val cross: (parents: Pair<T, T>) -> T,  

5      val mutate: (individual: T) -> T,  

6      val select: (scoredPopulation: Collection<Pair<Double, T>>) -> T) {  

7  

8      /**  

9          * @return O melhor individuo, após o número de epochs ter  

10         executado sob a população dada.  

11  

12         * @param epochs - numero de epochs (ciclos de mutação)  

13         * @param mutationProbability - valor entre 0 e 1 que define  

14         a probabilidade do Filho ser mutado  

15  

16         */  

17     fun run(epochs: Int = 1000, mutationProbability: Double = 0.1): T {  

18         var scoredPopulation = population.map { Pair(score(it), it) }.sortedByDescending {  

19             it.first }  

20  

21         for (i in 0..epochs)  

22             scoredPopulation = scoredPopulation  

23                 .map { Pair(select(scoredPopulation), select(scoredPopulation)) }  

24                 .map { cross(it) }  

25                 .map { if (random() <= mutationProbability) mutate(it) else it }  

26                 .map { Pair(score(it), it) }  

27                 .sortedByDescending { it.first }  

28  

29         return scoredPopulation.first().second  

30     }  

31 }  

32  

33 fun sudokuSolver(template : String) : List<List<Cell>> {

```

```

31  val boardTemplate = getBoardFromTemplate(template)
32  println("Initial Board: ")
33  boardTemplate.print()
34  println()
35  println()
36  val population = (1 .. 100).map {boardTemplate.fillWithRandValues().board}
37
38  val algorithm = GA<List<List<Cell>>>>(
39      population,
40      score = {81 - Board(it).getTotalCost().toDouble()},
41      cross = ::sudokuCrossover,
42      mutate = ::sudokuMutation,
43      select = ::sudokuSelectionTournament
44  )
45  return algorithm.run()
46  }
47  -
48  fun main(){
49
50      //val result = binaryArrayExample()
51      val result = sudokuSolver("
          .5..83.17...1..4..3.4..56.8....3...9.9.8245....6....7...9....5...729..86
          ")
52
53      print("Best individual: ")
54      result.forEach { print(it) }
55  }

```

---

Listagem 1.14: Algoritmo GA

```

+-----+-----+-----+
| 1 5 7 | 9 4 6 | 8 3 2 |
| 2 8 4 | 5 7 3 | 1 9 6 |
| 3 9 6 | 1 2 8 | 7 4 5 |
+-----+-----+-----+
| 6 7 2 | 3 5 1 | 9 8 4 |
| 5 4 9 | 2 8 7 | 6 1 3 |
| 8 3 1 | 4 6 9 | 2 5 7 |
+-----+-----+-----+
| 4 1 5 | 6 9 2 | 3 7 8 |
| 7 6 3 | 8 1 5 | 4 2 9 |
| 9 2 8 | 7 3 4 | 5 6 1 |
+-----+-----+-----+
Errors: 0
Execution time = 15697 ms

Process finished with exit code 0

```

(a) Fácil

```

+-----+-----+-----+
| 5 3 8 | 6 7 4 | 2 1 9 |
| 1 6 2 | 9 5 8 | 7 3 4 |
| 4 7 9 | 3 1 2 | 6 8 5 |
+-----+-----+-----+
| 2 5 4 | 8 6 3 | 1 9 7 |
| 9 1 3 | 2 4 7 | 5 6 8 |
| 6 8 7 | 1 9 5 | 3 4 2 |
+-----+-----+-----+
| 3 4 5 | 7 8 1 | 9 2 6 |
| 7 2 6 | 4 3 9 | 8 5 1 |
| 8 9 1 | 5 2 6 | 4 7 3 |
+-----+-----+-----+
Errors: 0
Execution time = 140901 ms

Process finished with exit code 0

```

(b) Média

```

+-----+-----+-----+
| 6 1 3 | 7 9 4 | 2 5 8 |
| 9 8 7 | 2 5 1 | 3 6 4 |
| 5 4 2 | 6 8 3 | 7 9 1 |
+-----+-----+-----+
| 3 5 6 | 9 4 2 | 8 1 7 |
| 8 7 4 | 5 1 6 | 9 3 2 |
| 1 2 9 | 3 7 8 | 5 4 6 |
+-----+-----+-----+
| 4 3 5 | 1 2 7 | 6 8 9 |
| 7 9 1 | 8 6 5 | 4 2 3 |
| 2 6 8 | 4 3 9 | 1 7 5 |
+-----+-----+-----+
Errors: 0
Execution time = 2825632 ms

Process finished with exit code 0

```

(c) Diabólico

Figura 1.6: Resultados GA

## 1.4 Conclusão

### 1.4.1 Resultados

Para cada algoritmo foi testado o seu tempo de resolução consoante 3 dificuldades diferentes de puzzle (Fácil, Médio, e Diabólico). Os Resultados permitem comparar a eficiência de resolução de cada algoritmo perante o mesmo tipo de puzzle, permitindo concluir que para os puzzles mais fáceis (Fácil e Médio), o algoritmo *A\** tem um menor tempo de resolução. Enquanto para os puzzles mais difíceis (Diabólico) o algoritmo *Simulated Annealing* tem um menor tempo de resolução.

Genetic Algorithm - GA parameters				
Difficulty	Pop. Size	Iterations	Mutation Prob.	Errors
Easy	500	2000	0,1	0
Medium	2000	8000	0,1	0
Diabolic	10000	10000	0,1	0

(a) Parâmetros do Algoritmo Genético

Performance Analysis - Sudoku Solver				
Language/Algorithm		Difficulty		
		Easy	Medium	Diabolical
SWI-Prolog	DFS	2443	45961	2 720 313
SWI-Prolog	A*	36	40	710848
Kotlin	SA	1406	3677	76464
Kotlin	GA	15697	140901	2825632
Note: Time expressed in millisecond				

(b) Análise de Performance

Figura 1.7: Resultados

### 1.4.2 Aprendizagem

Este trabalho prático apelou à nossa capacidade de implementar 4 algoritmos diferentes tanto em *Prolog* como em *Kotlin*. A observação da capacidade de resolução de cada

algoritmo perante um problema como um jogo de *Sudoku* permitiu-nos concluir tanto a eficiência como o próprio procedimento único que cada um destes algoritmos segue para chegar a um objetivo final.





# Referências

- [1] Wolfgang Ertel, *Introduction to Artificial Intelligence (Undergraduate Topics in Computer Science), Global Edition 4th ed.* Springer, 2017.
- [2] El-Ghazali Talbi, *Metaheuristics - From Design to Implementation*. Pearson, 2009.
- [3] Ivan Bratko, *Prolog Programming for Artificial Intelligence 4th ed.* Addison-Wesley, 2011.

